Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday @ 11:59 pm
  - Lab 1: next Monday @ 11:59 pm
  - Assignment 1: Monday Oct 10th @ 11:59 pm
- Lecture recordings are available on Canvas under Panopto Video
- Please use the “Request Regrade” feature on GradeScope if you have any issues with your grades
- TAs student support hours available on the syllabus page

Previous lecture

- Binary Search Tree
  - How to search, add, and delete
- Runtime of BST operations

Muddiest Points

- **Q: What's the difference between a binary tree and a regular tree**
- A: In a binary tree, each node has at most two nodes. There is also the notion of ordering the children of a node into a left child and a right child. In a general tree, the number of children is not limited and there is no specific ordering of a node's children.
- **Q: If we see a duplicate value in a data set that will be going into a BST, we can just ignore it since it was already added?**
- A: If the data items are of a primitive type (e.g., int, double, char), you can just ignore the duplicate. However, if the data items are objects of a reference type, it is possible to have two objects that are equal in a subset of the instance variables and different in others. In that case, we need to add the new object and return the replaced object.
- **Q: Why are recursive methods private? Why does it matter to hide them in a wrapper?**
- A: Recursion is an implementation detail. We don't want to change the client code if we decide to switch from a recursive implementation to an iterative implementation, for example. Also, calling recursive methods may be too complicated for the client code.

Muddiest Points

- **Q: At first I didn’t realize that the constraint for bst implies every single subtree but now it makes sense**
- A: Thank you for sharing the reflection
- **Q: Is no duplicates embedded into the 'national' definition of binary search tree or just for this class**
- A: No duplicates is both a simplifying assumption and an implication of storing (*key*, value) pairs in tree nodes. Not sure if there is a `national’ definition of BST.

Muddiest Points

- **Q: height vs depth and root's role in that**
- **Q: Is the height of the root node 1 or is it also 0 like the depth of it?**
- A: The height ***of a tree*** is the number of levels of the tree. The depth ***of a node*** is the number of edges from the root to the node. Root node’s depth is 0. Height of root is not defined.
- The height of a tree =

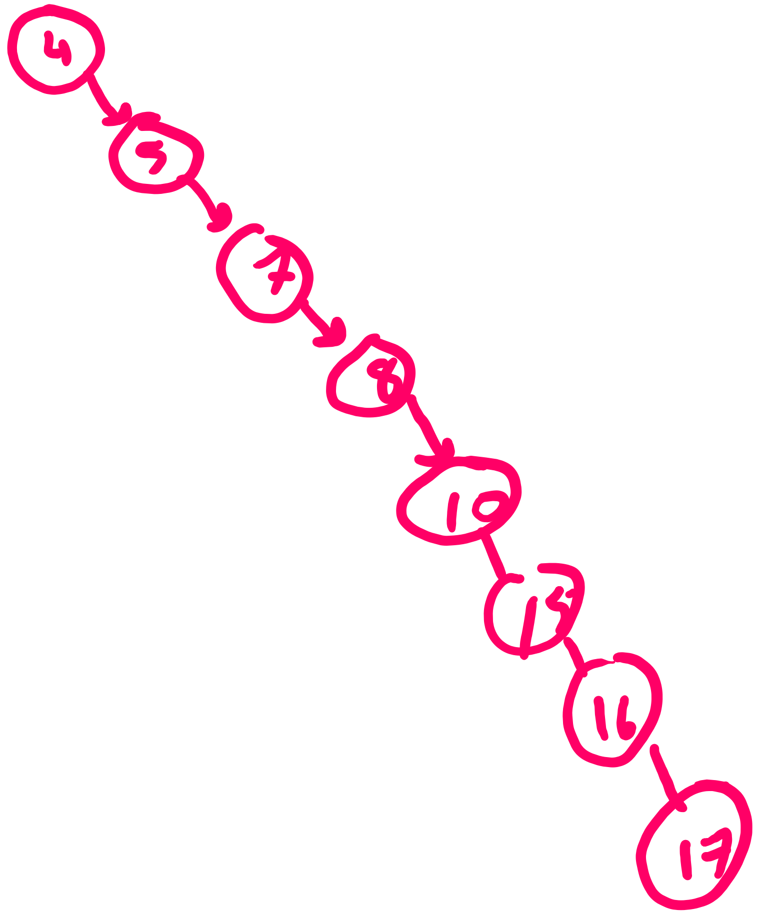
  1 + the largest depth of any node in the tree

Muddiest Points

- **Q: I've heard of rebalancing a binary tree. what does that mean?**
- A: It means maintaining a limit on the difference between left subtree’s height and right subtree’s height. We will learn about one way of rebalancing today.

Muddiest Points

- **Q: why do we compare 8 times before we add 20? Do we need to compare the first number?**
- A: Yes. Also, note that the second 8 in the input replaces the existing 8.



Muddiest Points

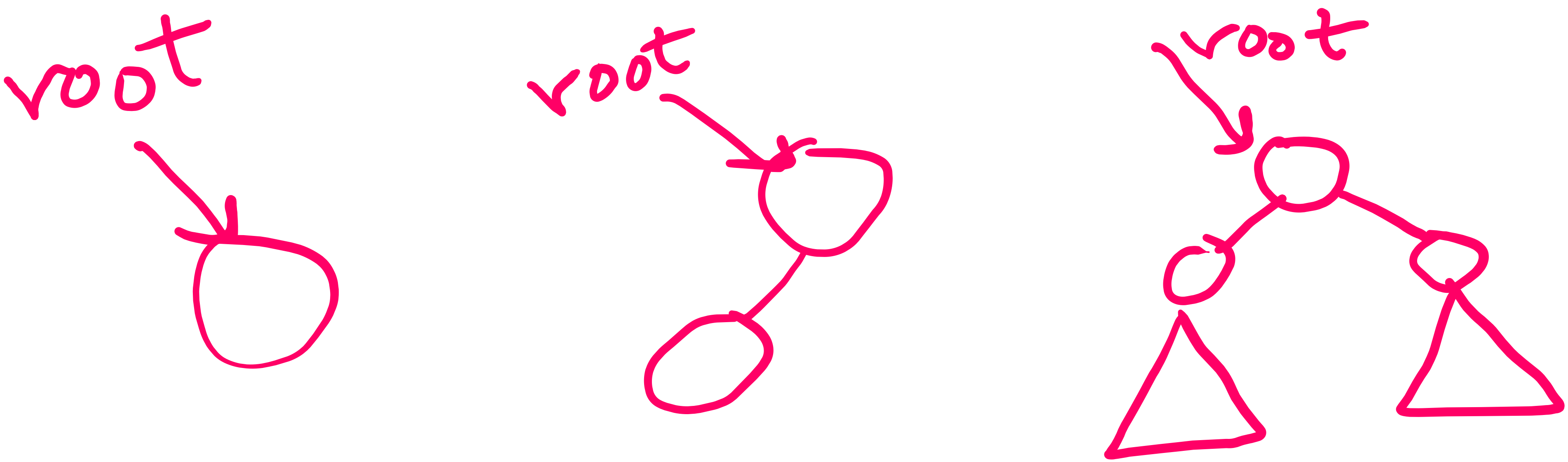
This Lecture

- Binary Search Tree
  - How to delete
- Runtime of BST operations
  - delete
- Red-Black BST (Balanced BST)
  - definition and basic operations

BST: delete operation

- Deleting an item requires first to find the node with that item in the tree
- Let's assume that we have already found that node
- The method below returns a reference to the root of the tree after removing its root

```
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



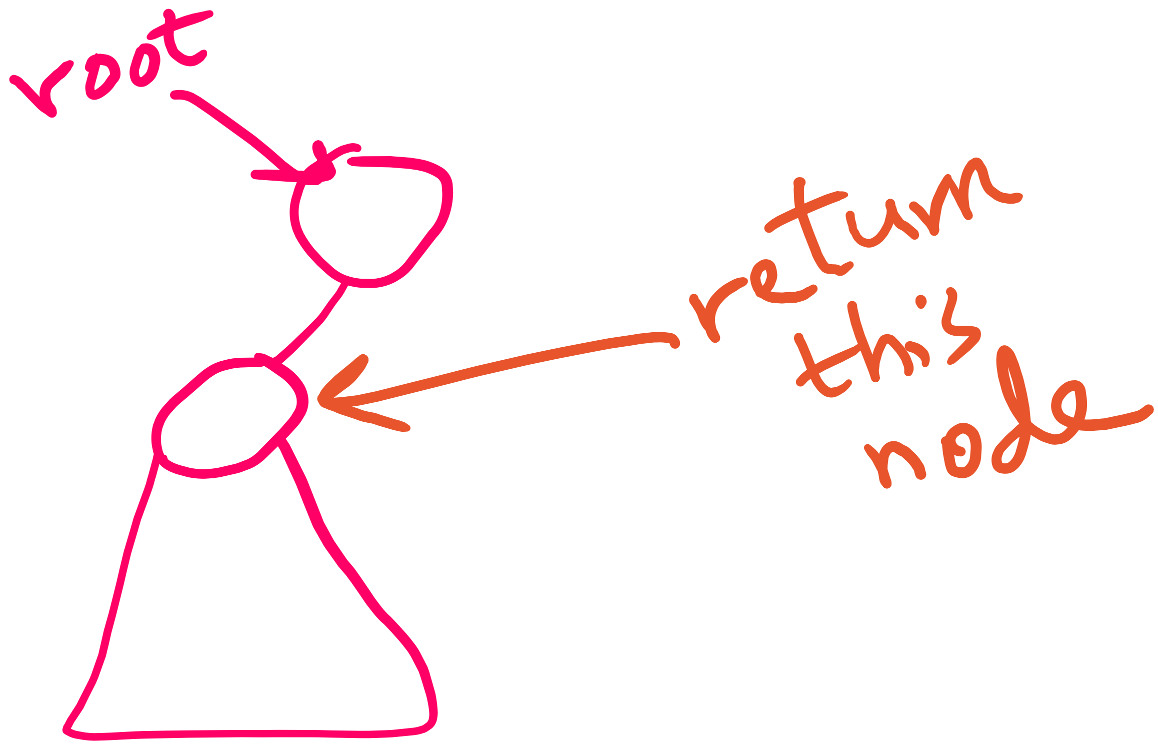
Delete Case 1: tree has only one node

```java
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```

Return null

Delete Case 1: root has one child (left or right)

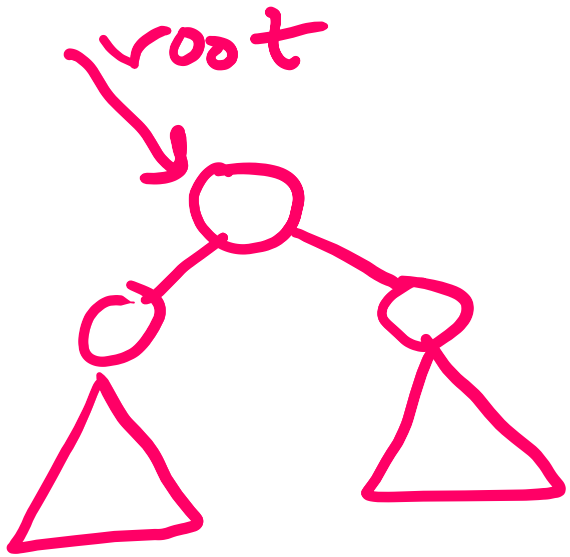
```java
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



Return the root of the subtree rooted at the child

Delete Case 1: root has two children

```java
private BinaryNode<T> removeFromRoot(BinaryNode<T> root){
```



- replace root's data by the data of the largest item of its left subtree (why?)
- remove the largest item from the left subtree
- return root

How to find largest item in a BST?

```java
private BinaryNode<T> findLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
    return findLargest(root.getRightChild());
  } else {
    return root;
  }
}
```

How to remove largest item in a BST?

- The method below returns the root of the tree after deleting the largest item
- If the largest item is the root of the tree, return its left child

```
private BinaryNode<T> removeLargest(BinaryNode<T> root){
  if(root.hasRightChild()){
     root.setRightChild(removeLargest(root.getRightChild()));
  } else {
    root = root.getLeftChild();
  }
  return root;
}
```

Now we need to find the node to delete

- The method below returns the root of the BST after removing the node that contains entry if found
- We also need to return the removed data item
  - How to return two things?
  - Pass a wrapper object

```
private BinaryNode<T> removeEntry(BinaryNode<T> root,
                                  T entry, ReturnObject item){
```

Wrapper Class

```
private class ReturnObject {
  T item;
  private ReturnObject(T entry){
    item = entry;
  }
  private void set(T entry){
    item = entry;
  }
  private T get(){
    return item;
  }
```

Runtime of BST operations

- Search miss, search hit, add
  - O(depth of node)
  - Worst-case: O(n)
  - Average-case: O(log n)
- Delete
  - Finding the node: O(log n) on average
  - Finding and removing largest node in subtree: O(log n) on average
  - Total is O(log n) on average
    - and O(n) in worst-case

Runtime of BST operations

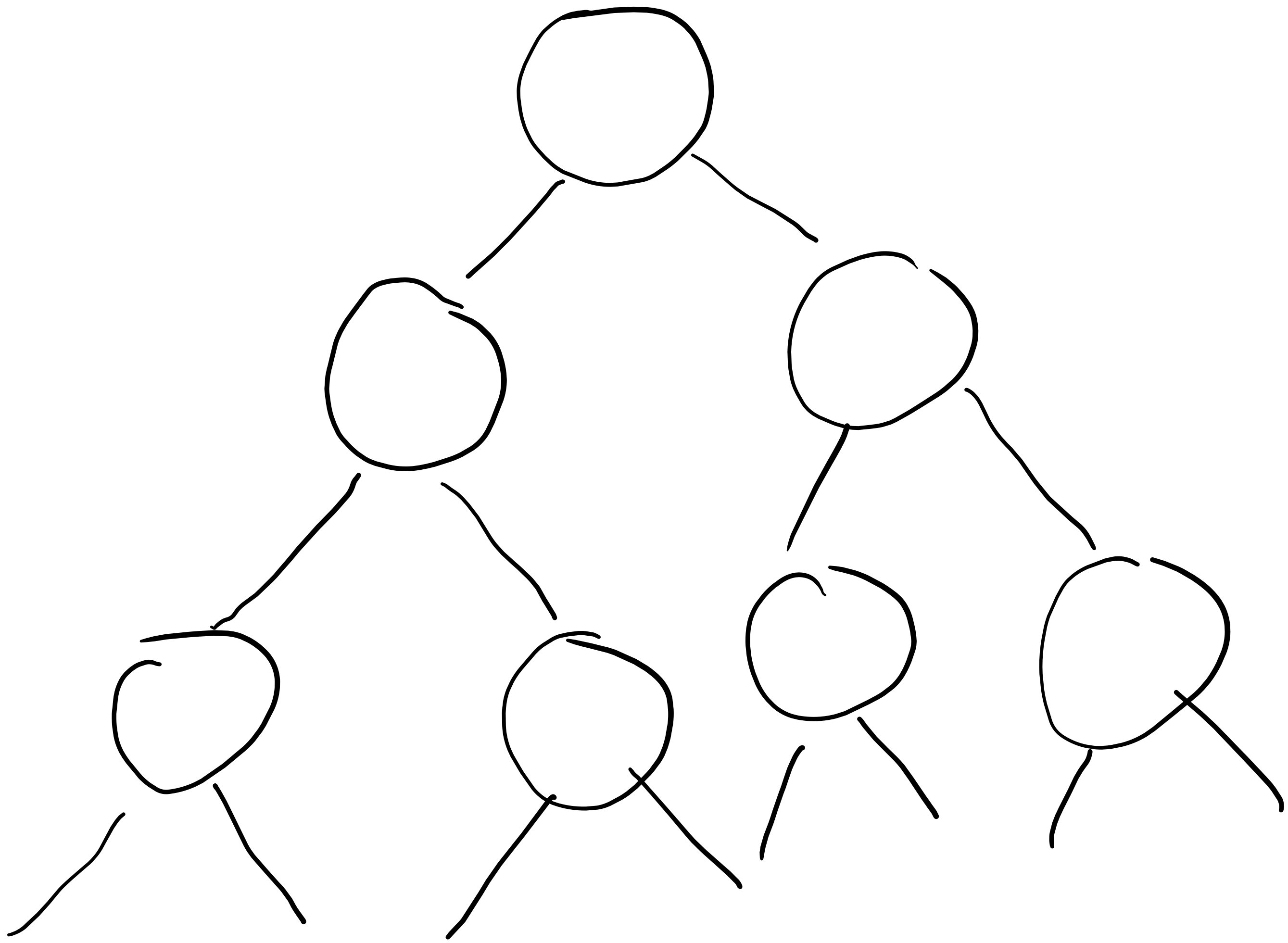
- Can we make the worst-case runtime O(log n)?
- Yes, if we keep the tree *balanced*
  - That is, the difference in height between left and right subtrees is controlled

Red-Black BST

- Definition
  - two colors for edges: red and black
  - a node takes the color of the edge to its parent
  - only left-child edges can be red
  - at most one red-edge connected to each node
  - Each leaf node has two black null-edges out of it (to the two null references)
  - all paths from root to null-edges have the same number of black edges
  - root node is black
  - ***Why?***
    - ***maximum** height = 2*log n*
- Basic operations
  - rotate left
  - rotate right
  - flip color
  - ***preserve the properties of the red-black BST!***

Red-black BST example

- All black nodes → has to be a full tree
- Height = O(log(n))

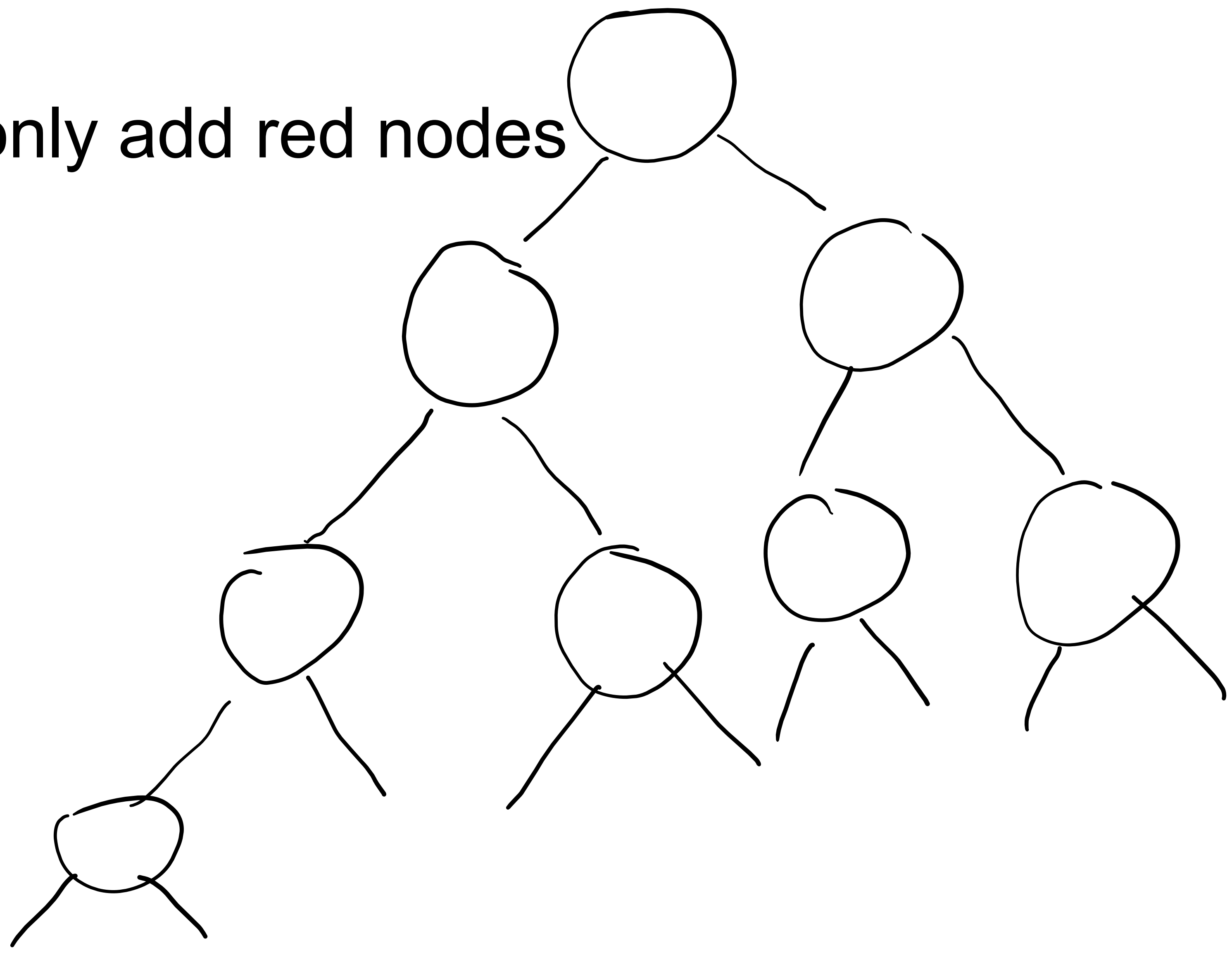


Red-Black BST example

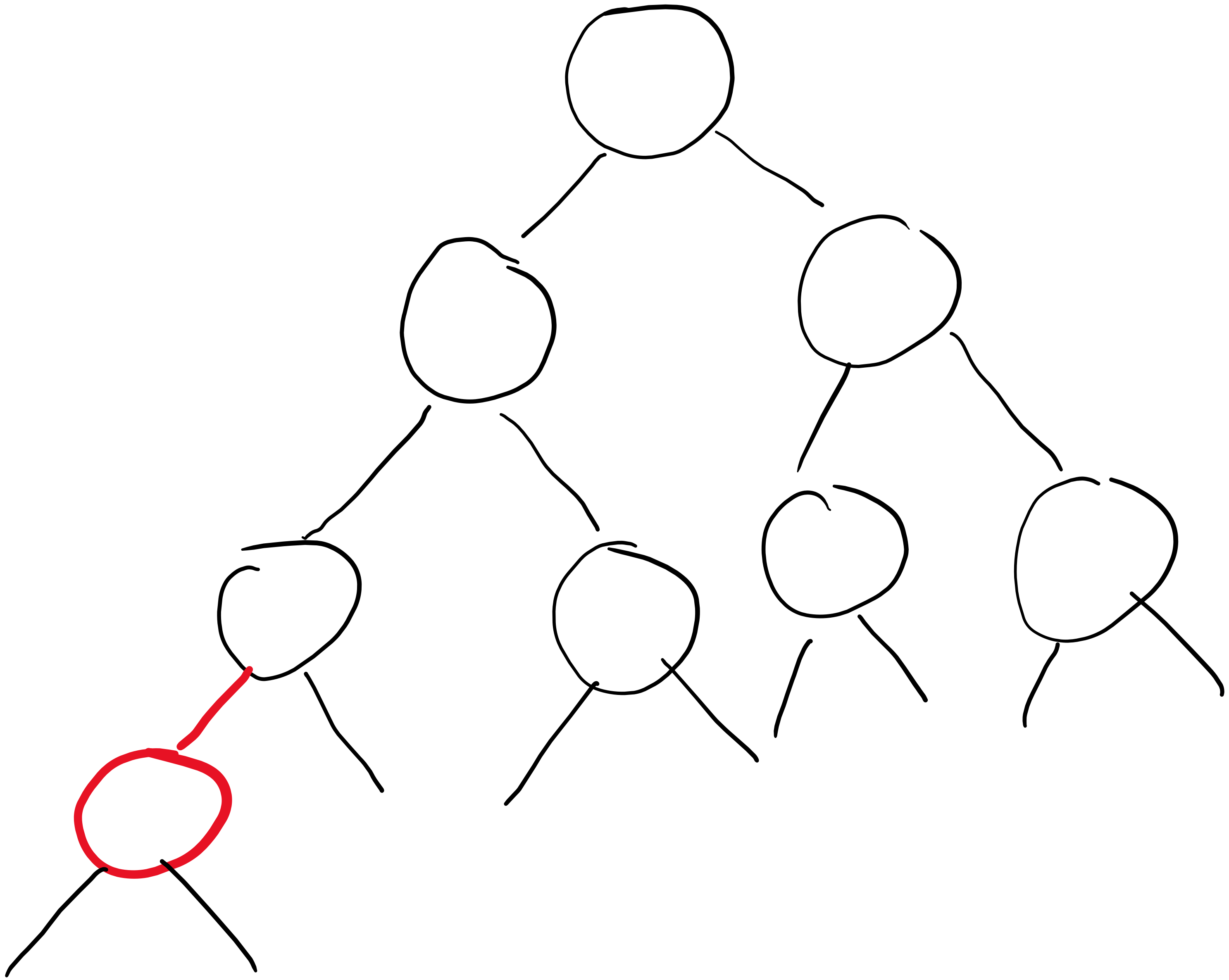
- Let's imagine an adversary who wants to increase the height of the tree by adding the fewest number of node

Red-black BST example

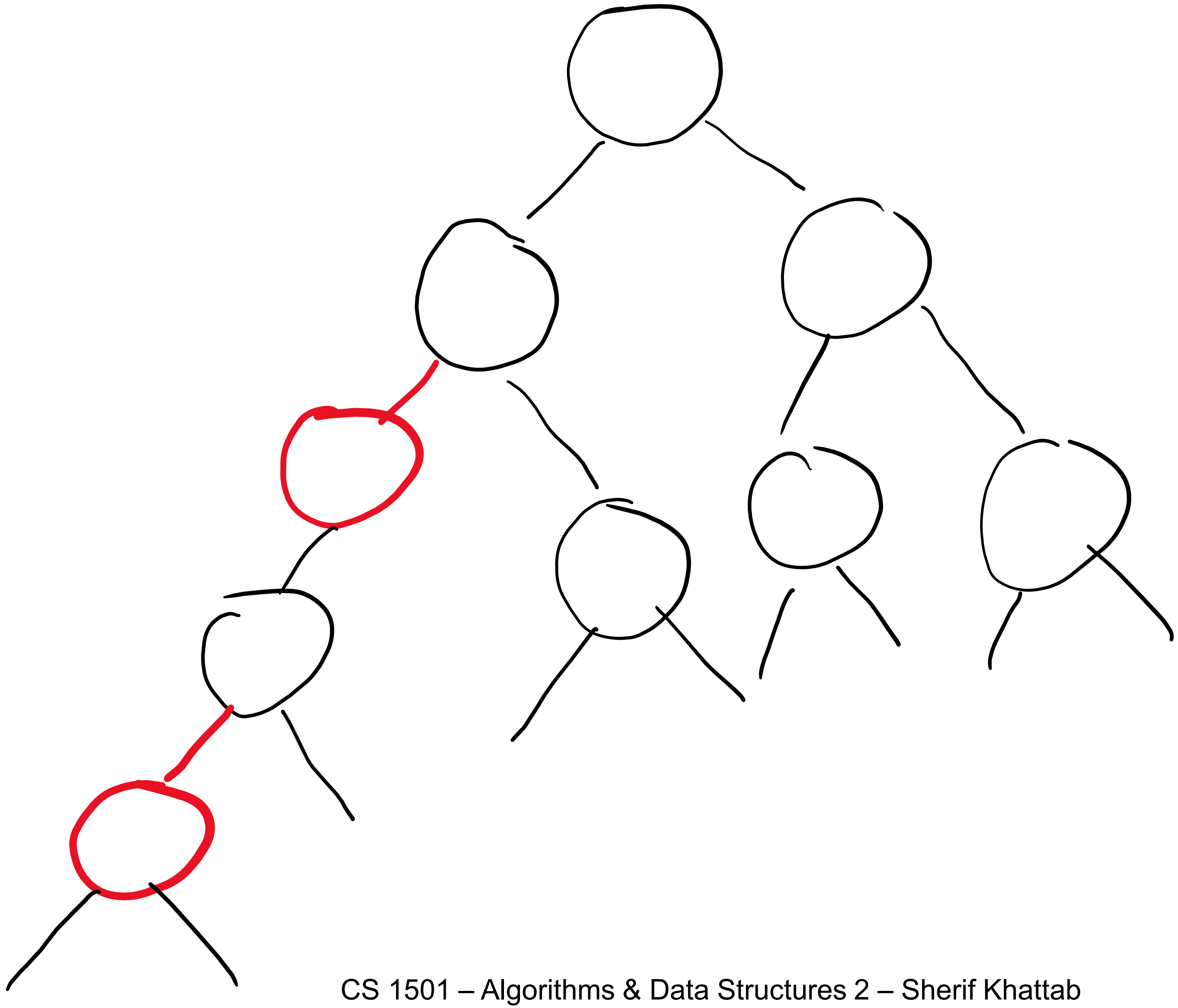
- Can the adversary add a black node?
- No! why?
- They can only add red nodes



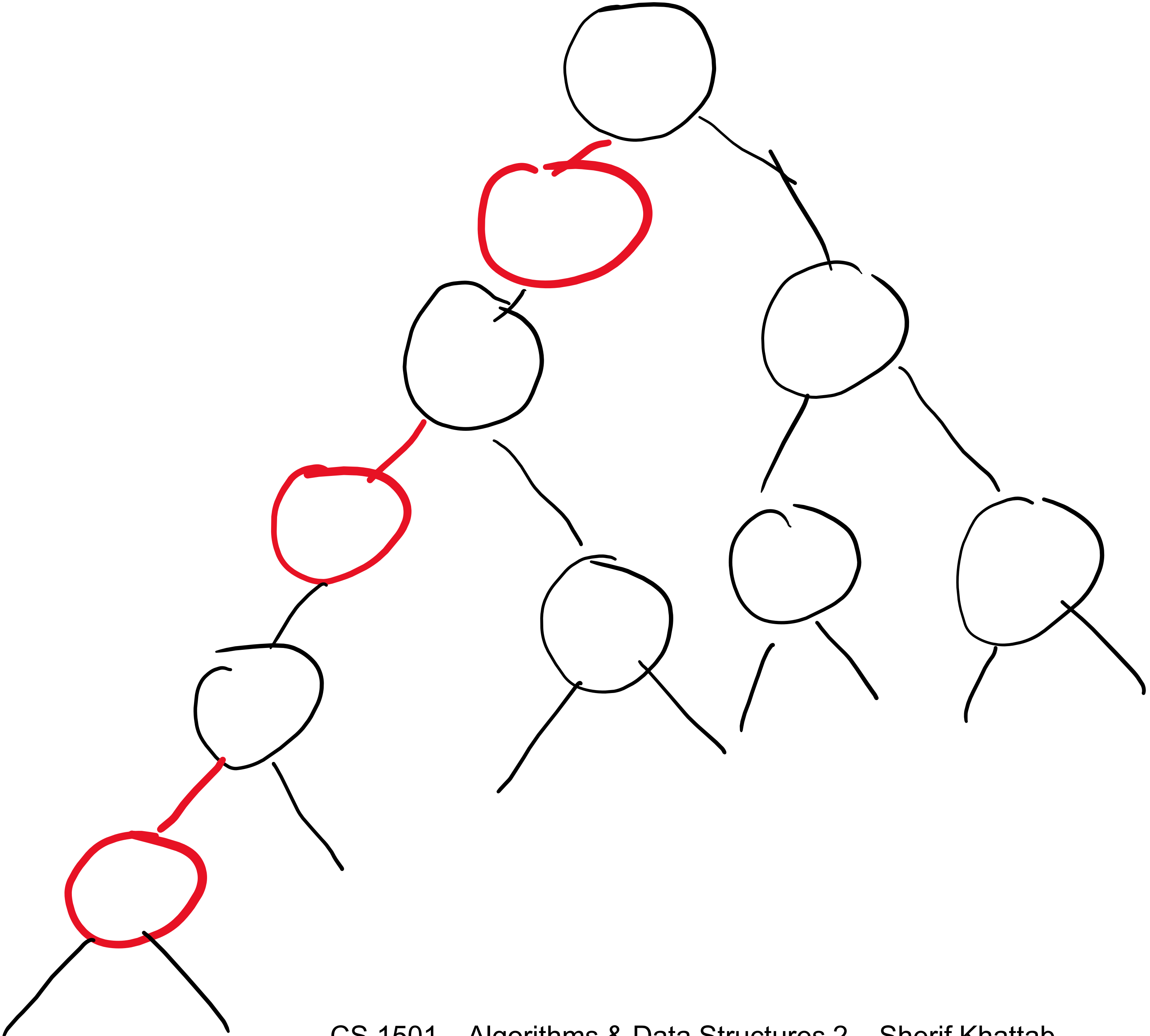
Red-black BST example



Red-black BST example

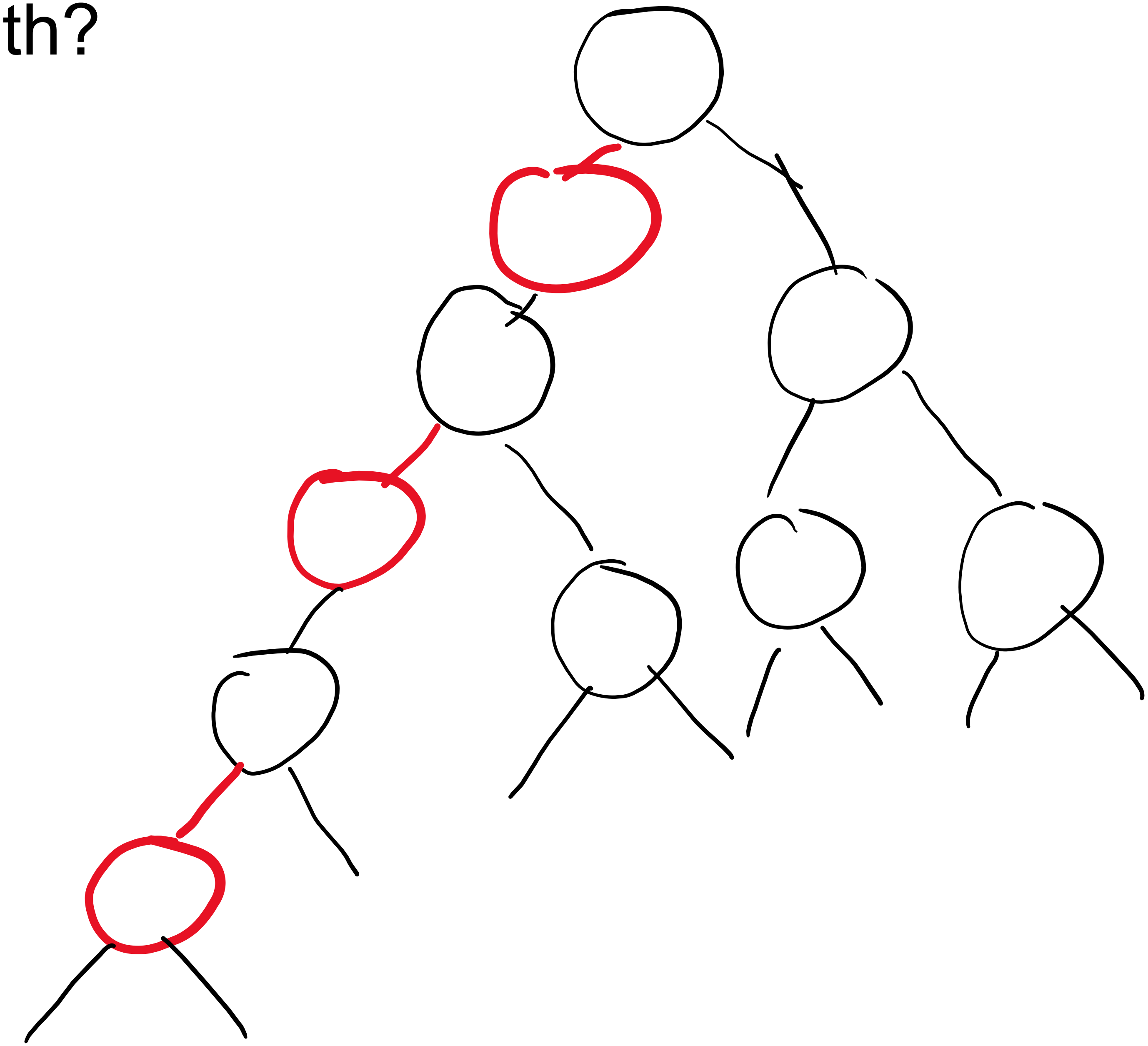


Red-black BST example



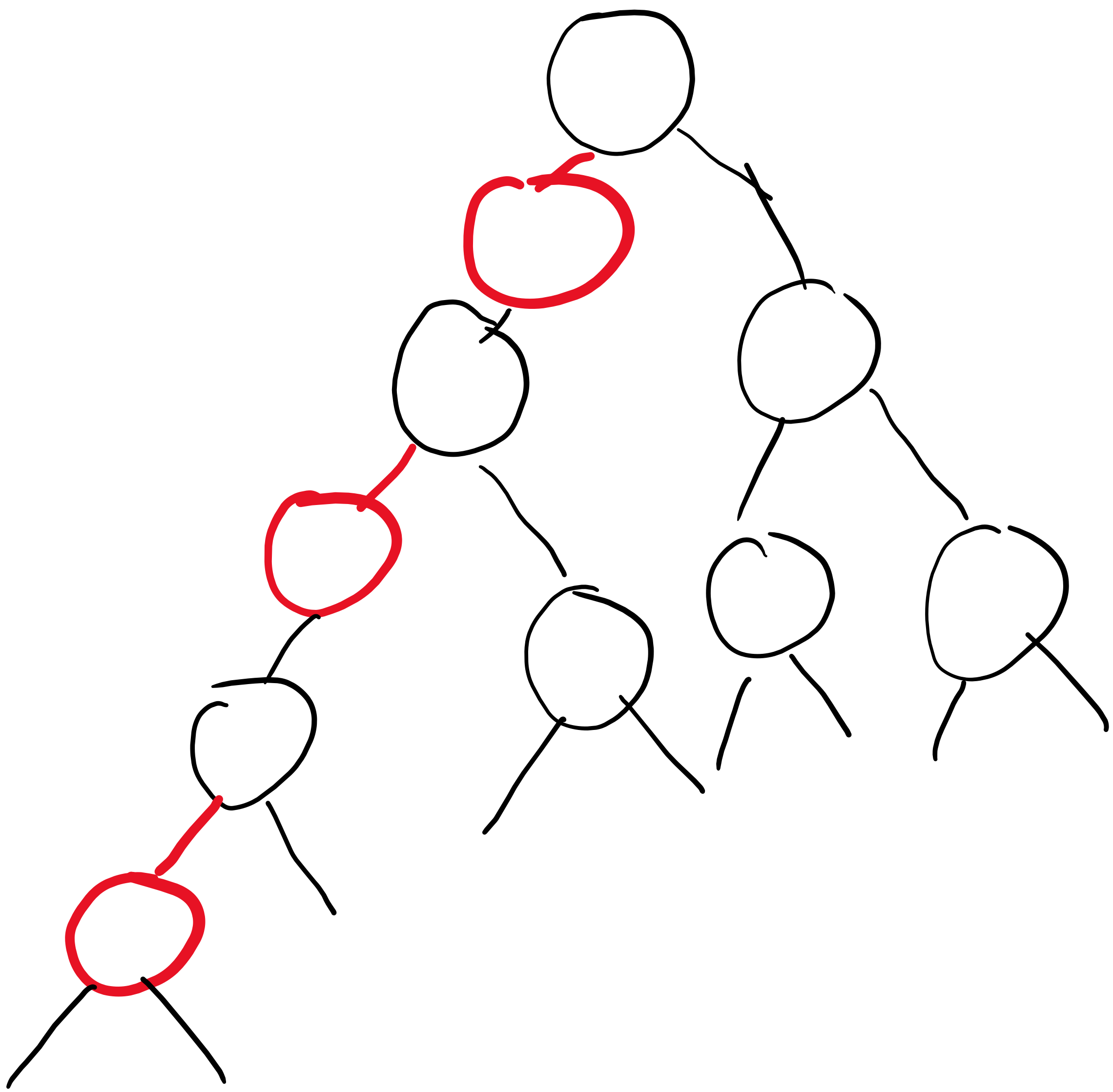
Red-black BST example

- Can the adversary add more red nodes to the left-most path?

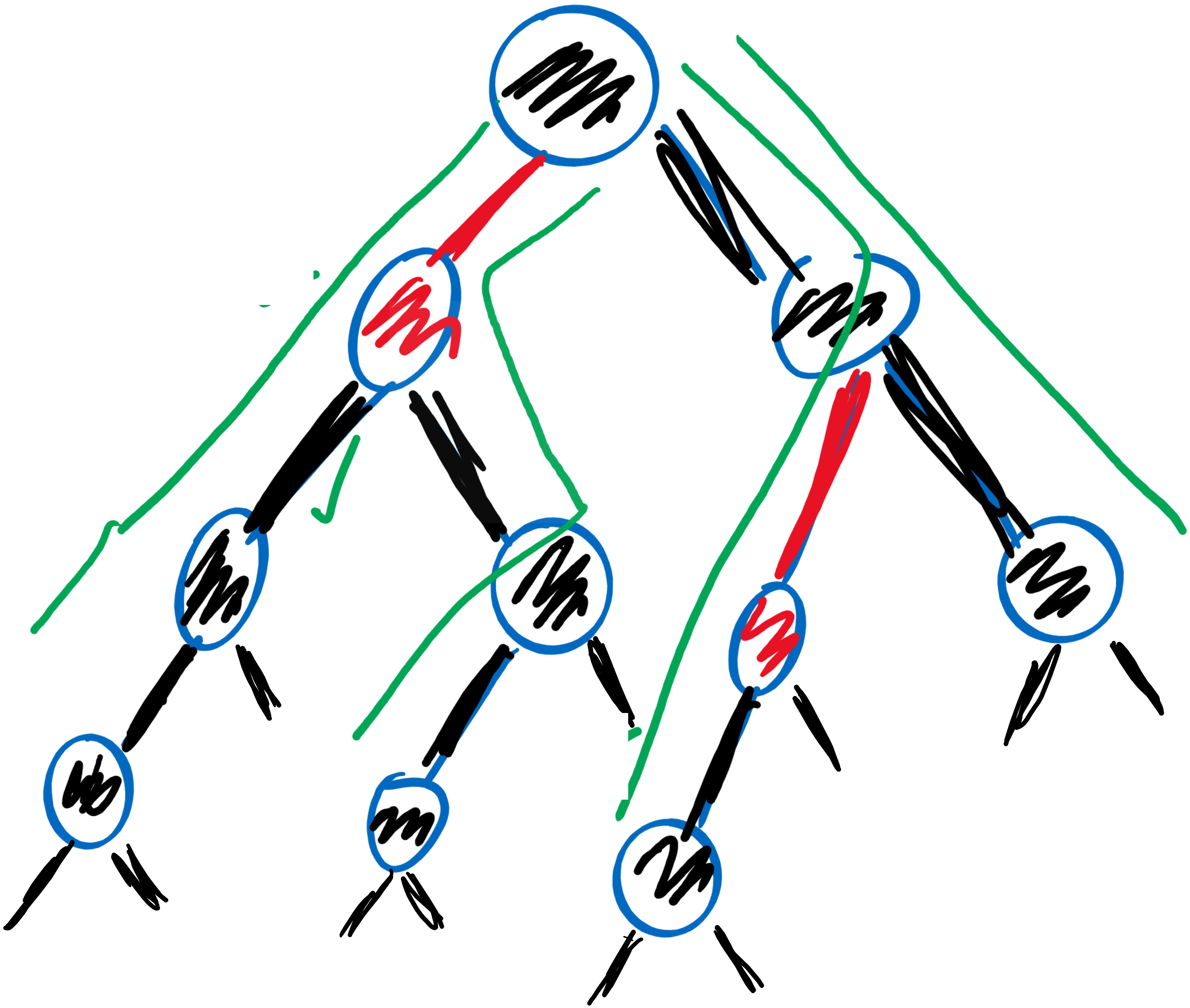


Red-Black BST Example

- The maximum "damage" that the adversary can do is to double the height of the full tree
  - 2* log (n)
  - still O(log n)



Red-black BST non-example

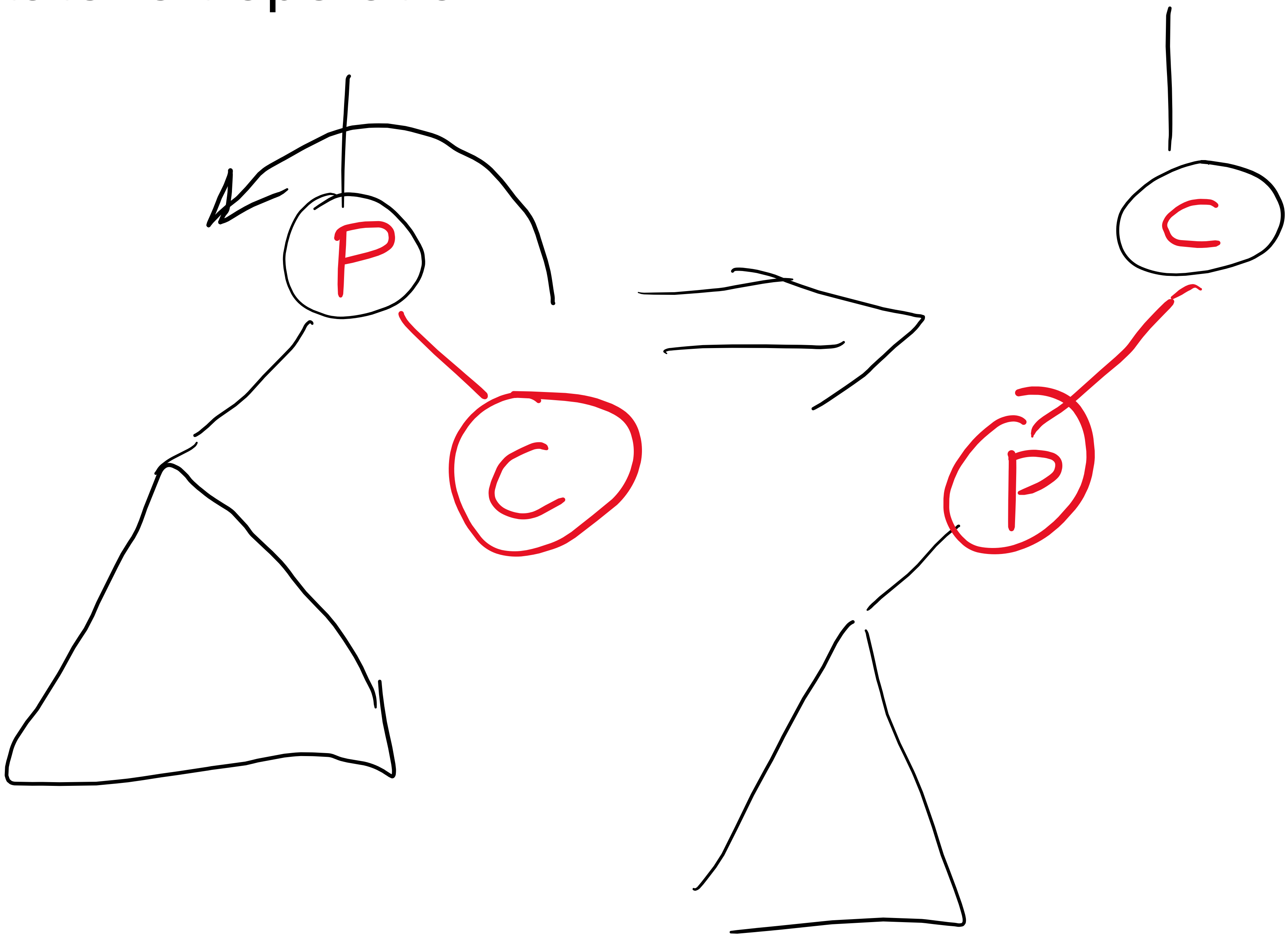


Adding to a RB-BST

- Ok, so we add a red leaf node!
- What can go wrong then?
  - The new node is a right child
  - The parent of the new node is also red
  - The sibling of the new node is also red

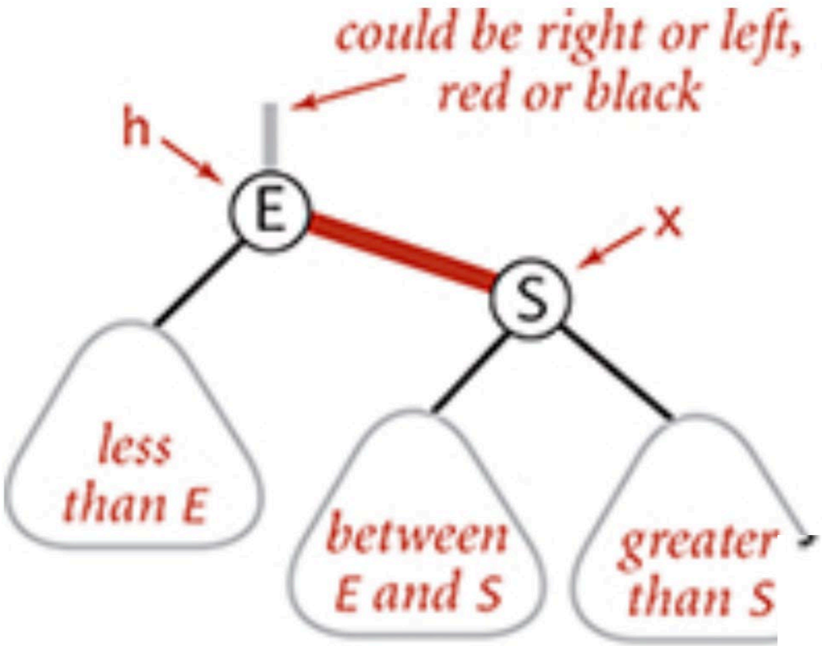
What if the new red node is a right child?

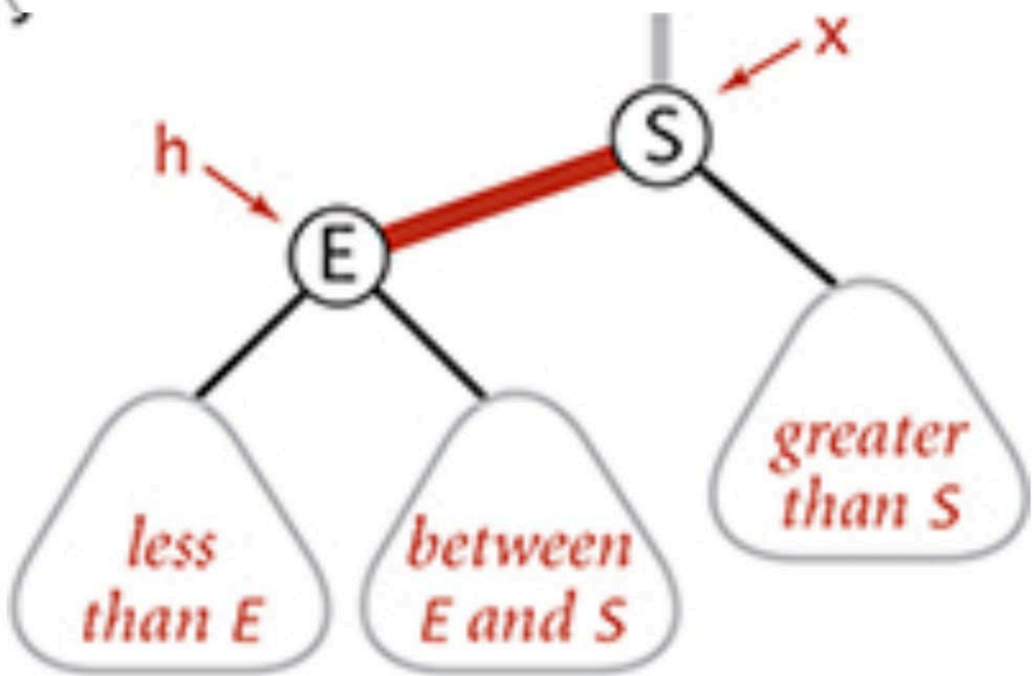
- rotateLeft operation



rotateLeft in general

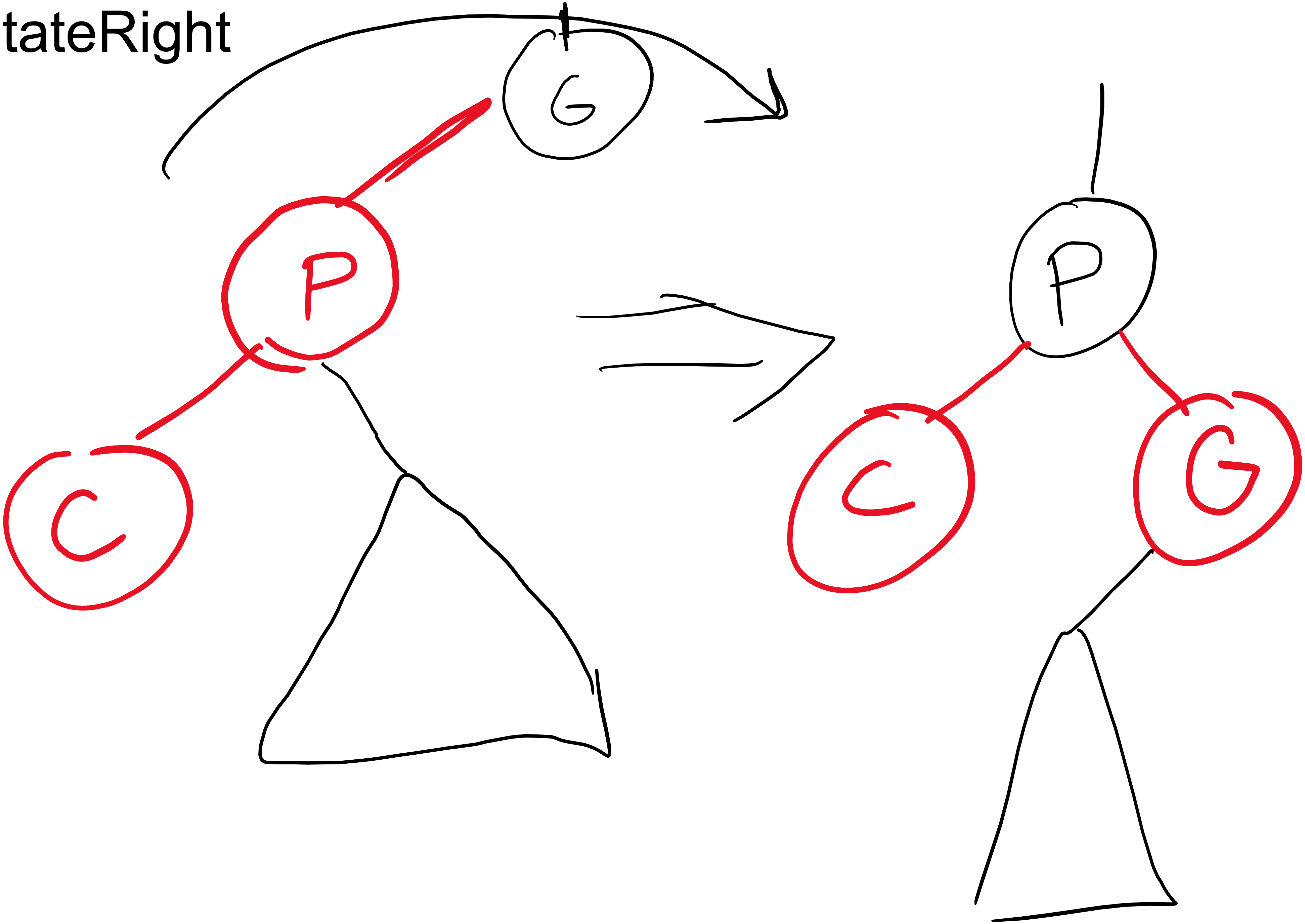
- rotateLeft operation



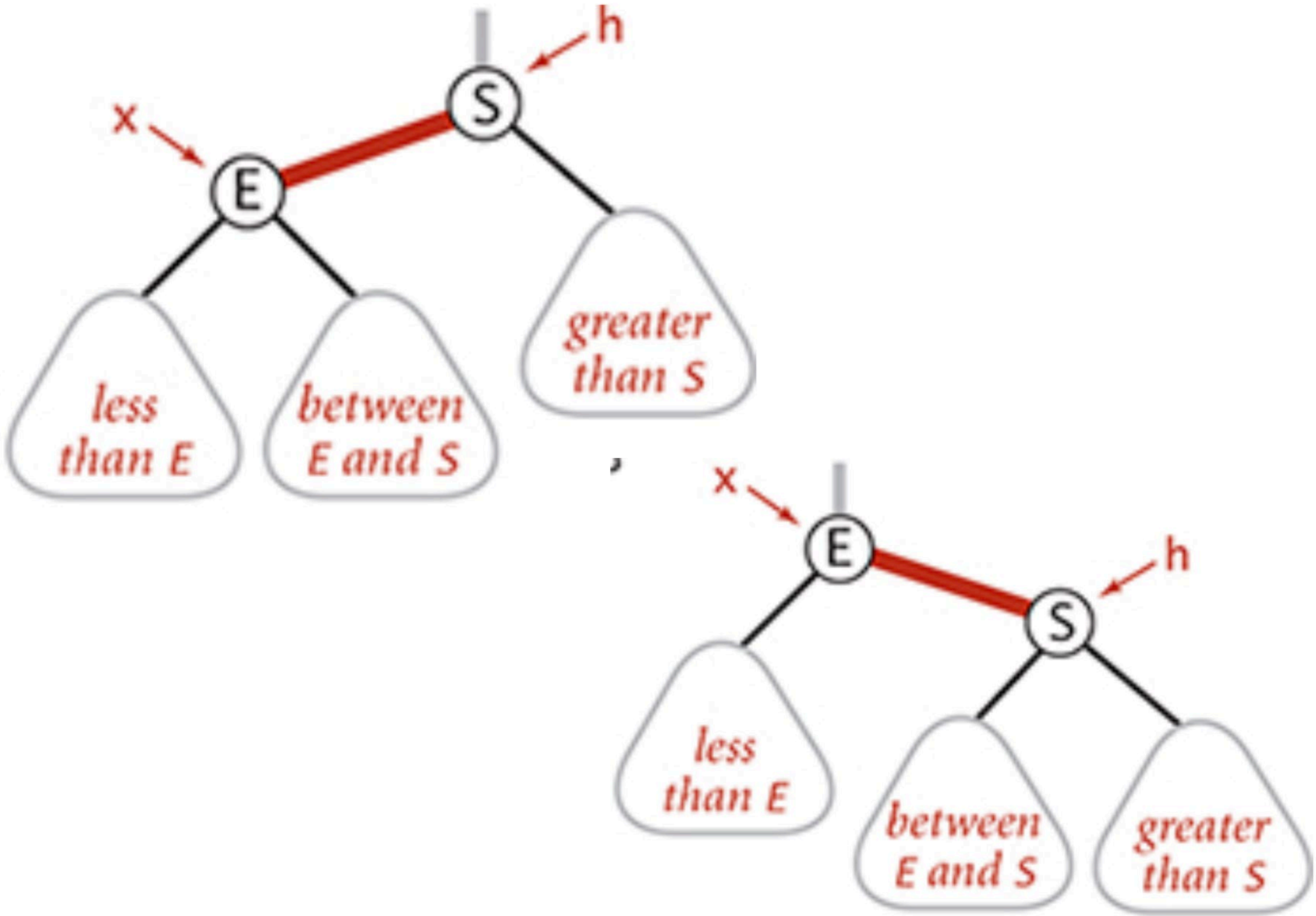


What if the parent of the new node also red?

- rotateRight

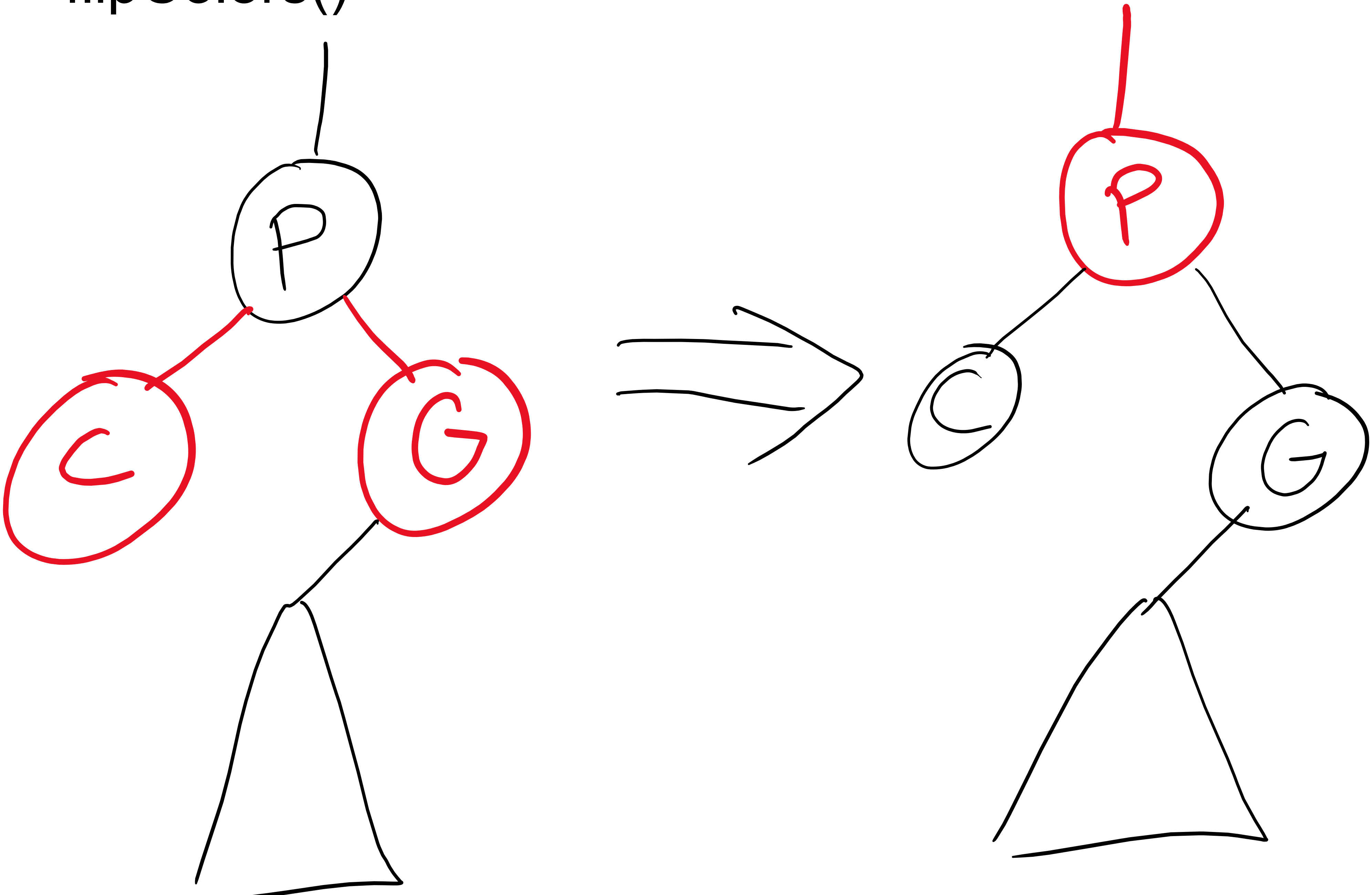


rotateRight



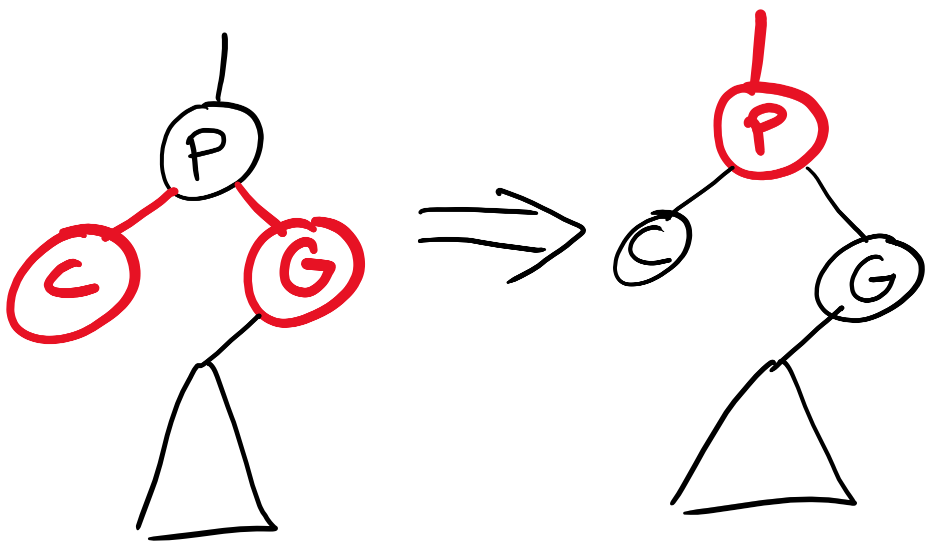
What if both children of a node are red?

- flipColors()



What if both children of a node are red?

- flipColors()
- Possible that changing P’s color to RED causes violations in the next level up!
- Need to correct the violations as we climb back up to root!
  - Correct violations after recursive call

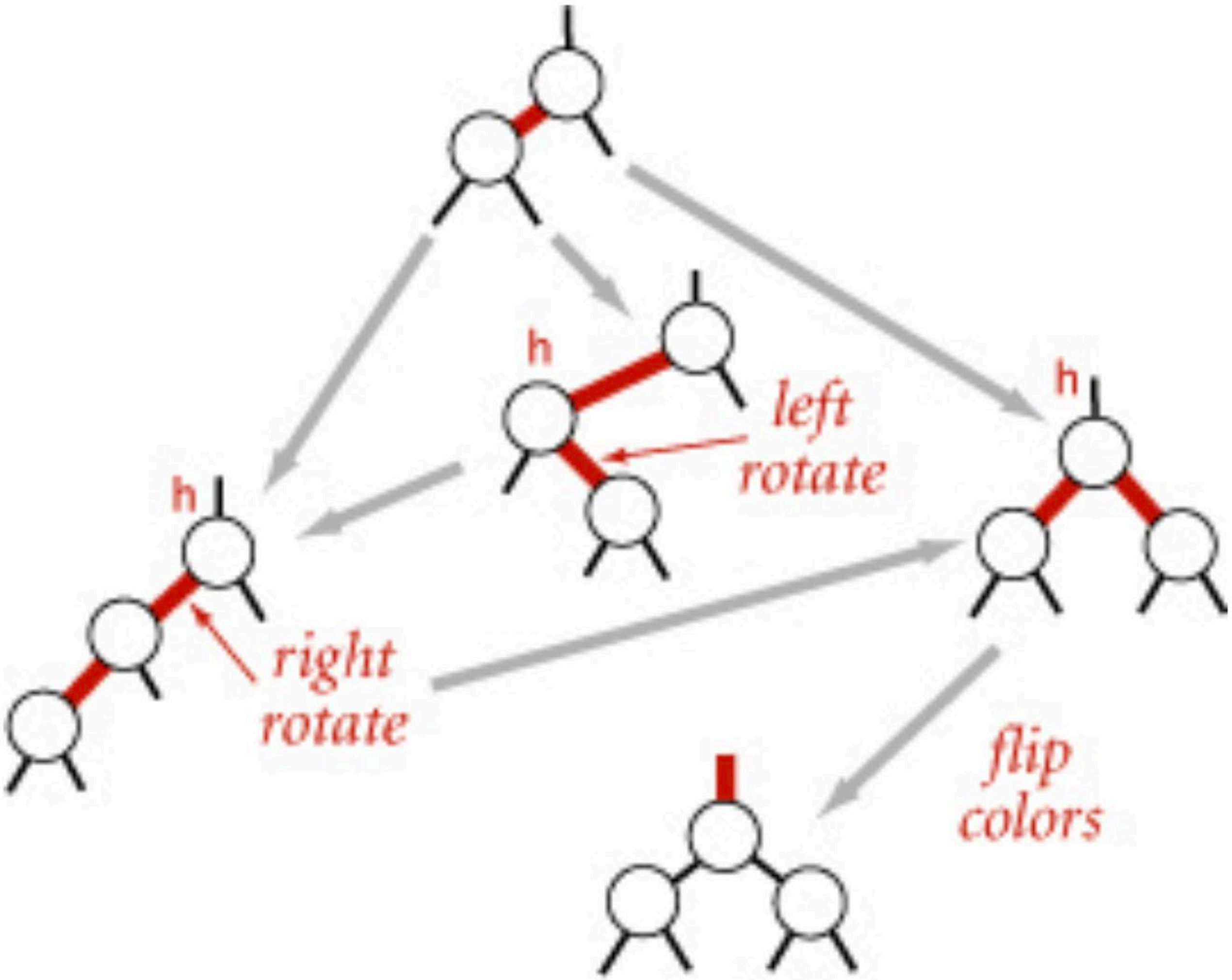


Adding to a red-black BST

- new node is always red (at least initially)
- if properties **violated**, correct using one or more of the basic operations
- Violations that can happen:
  - red link to the right child
  - two red links connected to the same node
- Correcting a violation may result in a violation up the tree
- Corrections happen as we climb back up the tree
  - That is, ***after the recursive call***
- If root node ends up to change to red, set it back to black

Which correction to do first?

- There are dependencies between corrections!



Which violations to check for first?

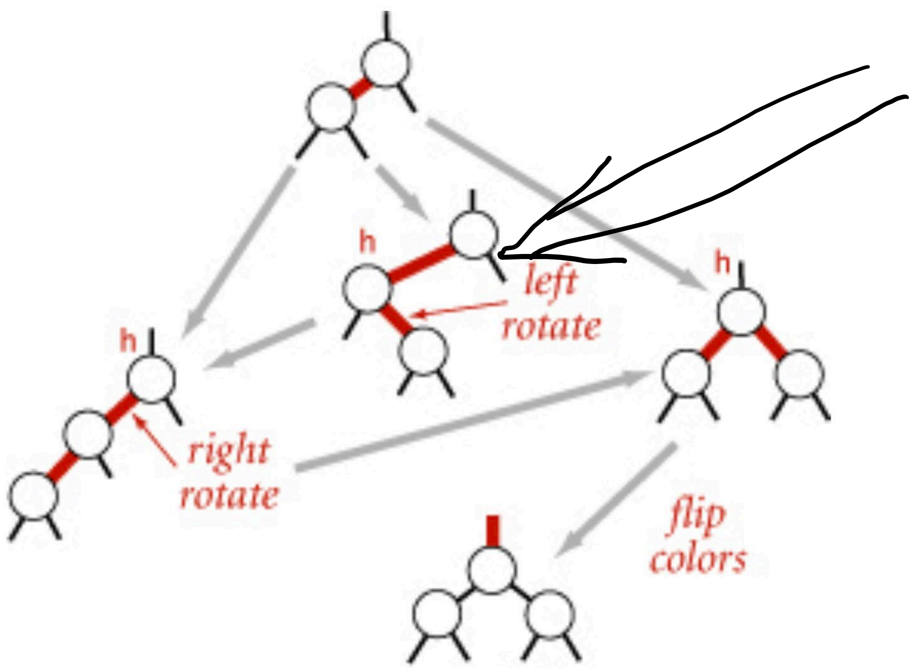
- TreeADT/RedBlackBST.java
- *h* starts as the parent of the new node and climbs up the tree

```java
// insert the key-value pair in the subtree rooted at h
    private Node put(Node h, Key key, Value val) {
        if (h == null) return new Node(key, val, RED, 1);

        int cmp = key.compareTo(h.key);
        if      (cmp < 0) h.left  = put(h.left,  key, val);
        else if (cmp > 0) h.right = put(h.right, key, val);
        else              h.val   = val;

        // fix-up any right-leaning links
        if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
        if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
        h.size = size(h.left) + size(h.right) + 1;

        return h;
    }
```

Which violations to check for first?

- TreeADT/RedBlackBST.java
- *h* starts as the parent of the new node and climbs up the tree

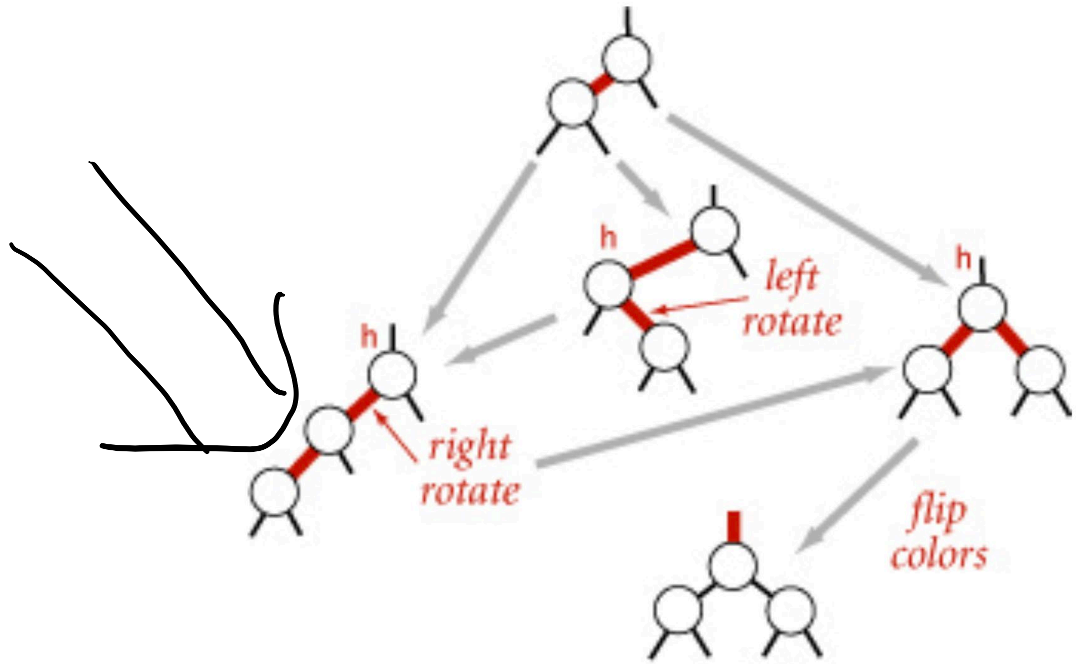
```
// fix-up any right-leaning links
if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
```



Which violations to check for first?

- TreeADT/RedBlackBST.java
- *h* starts as the parent of the new node and climbs up the tree

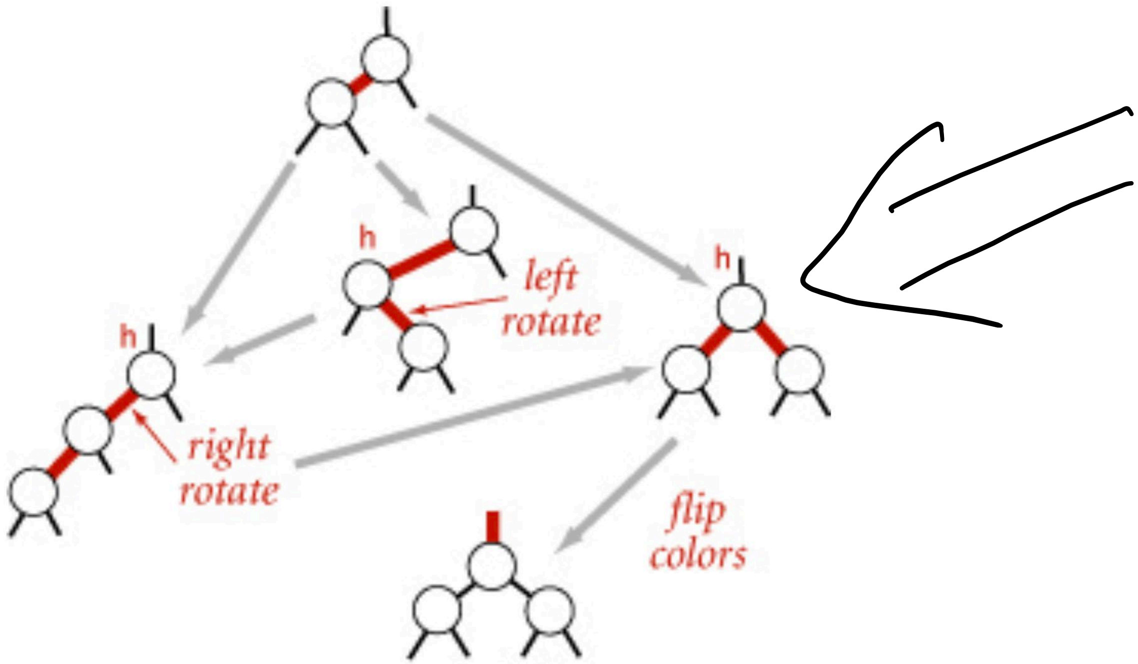
```
if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
```



Which violations to check for first?

- TreeADT/RedBlackBST.java
- *h* starts as the parent of the new node and climbs up the tree

```
if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
```



Deleting a node

- Make sure that we are not deleting a black node
  - as we go down the tree, make sure that the next node down is red
    - using a different set of operations
  - as we go back up the tree, correct any violations
    - same as we did while adding
  - if deleting a node with 2 children
    - replace with minimum of right subtree
    - delete minimum of right subtree
    - similar trick to delete in regular BST

Other BST operations

- Find successor and predecessor of an item
    - Lab 3
- Find all items within a specific range
    - Please check the keys methods in RedBlackBST.java inside the TreeADT folder in the code handouts
- Same code as regular BST!
- **worst-case runtime = Theta(log n)**